Improving Abstract Interpretations by Combining Domains

MICHAEL CODISH
Ben-Gurion University of the Negev
and
ANNE MULKERS and MAURICE BRUYNOOGHE
Katholieke Universiteit Leuven
and
MARIA GARCÍA DE LA BANDA and MANUEL HERMENEGILDO
Universidad Politécnica de Madrid

This article considers static analysis based on abstract interpretation of logic programs over combined domains. It is known that analyses over combined domains provide more information potentially than obtained by the independent analyses. However, the construction of a combined analysis often requires redefining the basic operations for the combined domain. A practical approach to maintain precision in combined analyses of logic programs which reuses the individual analyses and does not redefine the basic operations is illustrated. The advantages of the approach are that (1) proofs of correctness for the new domains are not required and (2) implementations can be reused. The approach is demonstrated by showing that a combined sharing analysis — constructed from "old" proposals — compares well with other "new" proposals suggested in recent literature both from the point of view of efficiency and accuracy.

General Terms: Languages, Performance

Additional Key Words and Phrases: Abstract interpretation, logic programming, program analysis

---

This research was supported in part by CEC DGXIII ESPRIT Project "PRINCE" and CICYT project TIC91-0106-CE. The work of M. Codish was supported by a postdoctoral fellowship from K.U. Leuven. M. Bruynooghe is supported by the Belgium National Fund for Scientific Research.
Authors' addresses: M. Codish, Department of Mathematics and Computer Science, Ben-Gurion University, P.O.B. 653, 84105 Beer-Sheba, Israel; email: codish@bengus.bgu.ac.il; A. Mulkers and M. Bruynooghe, Department of Computer Science, Katholieke Universiteit Leuven, Heverlee, Belgium; email {anne;maurice}@cs.kuleuven.ac.be; M. García de la Banda and M. Hermenegildo, Universidad Politécnica de Madrid, Facultad de Informática, Madrid, Spain; email: {maria;herme}@fi.upm.es.

1. INTRODUCTION

The framework of abstract interpretation [Cousot and Cousot 1977] provides the basis for a semantic approach to dataflow analysis. A program analysis is viewed as a nonstandard, abstract semantics defined over a domain of data descriptions. An abstract semantics is constructed by replacing operations in a suitable concrete semantics with corresponding abstract operations defined on data descriptions. Program analyses are defined by providing finitely computable abstract interpretations which preserve interesting aspects of program behavior.

Describing program analysis as a nonstandard semantics is more than a theoretical exercise in aesthetics. The semantic approach allows us to focus on the abstraction of data. The framework of abstract interpretation then determines an abstract semantic domain and an abstract semantics. Formal justification of program analyses is reduced to proving conditions on the relation between data and data descriptions and on the elementary operations defined on the data descriptions. This approach eases both the development and the justification of program analyses.

In the case of logic programming languages, "data" corresponds to substitutions and atoms. The basic operations on data typically include unification, composition of substitutions, and projection of substitutions onto variables of interest. Proving the safety of an abstract unification function is the major step in proving the safety of abstractions for logic programs. Introductory material for the subject of abstract interpretation of logic programs can be found for example in Debray [1992], Cousot and Cousot [1992], Jones and Søndergaard [1987], and Bruynooghe and Boulanger [1994].

It is often the case that program analyses aim to provide a combination of different types of information. Typical examples in the context of logic programs are analyses for: groundness and sharing [Codish et al. 1991; Jacobs and Langen 1992; Muthukumar and Hermenegildo 1992; Søndergaard 1986], modes and types [Janssens and Bruynooghe 1992; Horiuchi 1992], sharing and freeness [Muthukumar and Hermenegildo 1991; Sundararajan and Conery 1992], etc. Typically, such combined analyses provide more information than that obtained by combining the results of the individual analyses. Moreover, efficiency can also improve as the increased precision reduces the number of irrelevant analysis paths which the abstract computation is obliged to follow. However, the design, implementation, and formal justification of combined analyses usually require new efforts which do not directly benefit from previously designed analyses.

In this article we observe that in many cases it is possible to provide combined analyses which benefit from previously defined analyses, maintain a high degree of precision, and improve the efficiency of analyses. In particular, this is the case when the analyses being composed contain a sufficient degree of "overlapping" information. For example, recent proposals to achieve better sharing analyses by combining together the advantages of various old analyses [Cortesi and Filé 1993; Sundararajan and Conery 1992] can be derived automatically with little effort.

The theoretical background for the current article was laid down by Cousot and Cousot [1979]. There, the authors illustrate that although some precision can be gained by removing redundancies from combined domains, still further precision is

gained by introducing new basic operations. Here, we focus first on the precision that can be gained simply by removing redundancies. We illustrate for the case of logic programs that this often provides a practical technique for providing precise combined analyses. We also propose and illustrate an approach which allows deriving more precise information by removing redundancies and combining lower-level operations in previously defined analyses.

The rest of the article is structured as follows. Section 2 reviews the theoretical background for our technique. This includes a brief introduction to abstract interpretation following Cousot and Cousot [1977] as well as the definitions for combining domains as given in Cousot and Cousot [1979]. We close Section 2 with an example for logic programs which demonstrates how Søndergaard's domain for (pair) sharing and groundness analysis [Søndergaard 1986] can be constructed by combining corresponding groundness and sharing domains. In Section 3 we introduce an alternative domain for this type of analysis proposed in Jacobs and Langen [1992]. We identify the advantages of the two alternative domains and propose to provide the best of both worlds by combining the analyses respectively described in Codish et al. [1991] and in Muthukumar and Hermenegildo [1992] for these domains. Another example involves combining the sharing analysis of Codish et al. [1991] with the Sharing+Freeness analysis presented in Muthukumar and Hermenegildo [1991] and Muthukumar et al. [1992]. Section 4 provides an experimental evaluation of our approach. The combined analyses have been implemented in the context of the &-Prolog compiler [Bueno et al. 1994; Hermenegildo and Greene 1990; Muthukumar and Hermenegildo 1991; 1992] by reusing components of previously defined analyses. The results obtained are at least as good as those obtained in recently developed analyzers which completely redesign and reimplement the basic operations. Finally, Section 5 concludes and proposes some directions for further experimentation. This article is a revised version of Codish et al. [1993].

2. BACKGROUND

In the following we summarize briefly the theory of abstract interpretation as defined in Cousot and Cousot [1977]. The theory for combining domains follows the description in Cousot and Cousot [1979].

Abstract Interpretation

We assume the standard framework of abstract interpretation as defined in terms of *Galois insertions.*

Definition 2.1. (*Galois Insertion*). A Galois insertion is a quadruple (E, α, D, γ) where:

(1) $(E, \sqsubseteq_E)$ and $(D, \sqsubseteq_D)$ are complete lattices called *concrete* and *abstract domains* respectively;
(2) $\alpha : E \to D$ and $\gamma : D \to E$ are monotonic functions called *abstraction* and *concretization functions* respectively; and
(3) $\alpha(\gamma(d)) = d$ and $e \sqsubseteq_E \gamma(\alpha(e))$ for every $d \in D$ and $e \in E$.

In practice it is sufficient to specify only γ (or α). In the following we adhere to this policy.

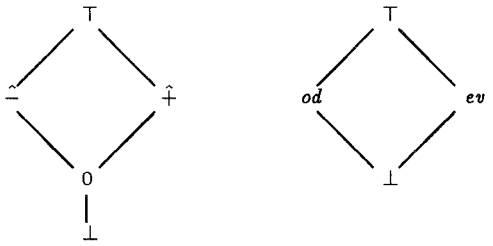

Fig. 1. *Sign* and *Parity* lattices.

Example 2.2. Let $Sign = \{ \bot, 0, \hat{+}, \hat{-}, \top \}$ and $Parity = \{ \bot, od, ev, \top \}$ be the complete lattices illustrated in Figure 1. Let

$\gamma_{sign} = \{ \bot \mapsto \emptyset,\ 0 \mapsto \{0\},\ \hat{+} \mapsto \{x \mid x \geq 0\}, \hat{-} \mapsto \{x \mid x \leq 0\},\ \top \mapsto \mathcal{Z} \}$;

$\gamma_{parity} = \{ \bot \mapsto \emptyset,\ ev \mapsto \{x \mid x \bmod 2 = 0\}, od \mapsto \{x \mid x \bmod 2 = 1\},\ \top \mapsto \mathcal{Z} \}$.

The following specifies the notion of approximation which is then lifted from the primitive domains to function domains:

Definition 2.3. (*Approximation*). Let (E, α, D, γ) be a Galois insertion, and let $\mu : E \to E$ and $\mu^{\mathcal{A}} : D \to D$ be monotonic functions. We say that $d \in D$ γ-*approximates* $e \in E$, denoted $d \propto_\gamma e$, if $e \sqsubseteq_E \gamma(d)$. We say that $\mu^{\mathcal{A}}$ γ-*approximates* μ, denoted $\mu^{\mathcal{A}} \propto_\gamma \mu$, if $\forall d \in D.\ e \in E.\ d \propto_\gamma e \Rightarrow \mu^{\mathcal{A}}(d) \propto_\gamma \mu(e)$.

Example 2.4. Consider the functions: $inc, dec, div : \mathcal{Z} \to \mathcal{Z}$ defined respectively by $\lambda x.x + 1$, $\lambda x.x - 1$, and $\lambda x.x$ div 2. Possible approximations for these functions over *Sign* and *Parity* are given by:

	$\bot$	0	$\hat{+}$	$\hat{-}$	$\top$
$inc_s^{\mathcal{A}}$	$\bot$	$\hat{+}$	$\hat{+}$	$\top$	$\top$
$dec_s^{\mathcal{A}}$	$\bot$	$\hat{-}$	$\top$	$\hat{-}$	$\top$
$div_s^{\mathcal{A}}$	$\bot$	0	$\hat{+}$	$\hat{-}$	$\top$

	$\bot$	*od*	*ev*	$\top$
$inc_p^{\mathcal{A}}$	$\bot$	*ev*	*od*	$\top$
$dec_p^{\mathcal{A}}$	$\bot$	*ev*	*od*	$\top$
$div_p^{\mathcal{A}}$	$\bot$	$\top$	$\top$	$\top$

Concrete semantics are typically defined as least fixed points of an operator on programs. Typically, the meaning of a program P may be expressed as $[\![P]\!] = lfp(f_P)$ where $f_P : Den \to Den$ is a monotonic operator on a domain of denotations Den. A program analysis will typically be defined by introducing an appropriate Galois insertion $(Den, \alpha, Den^{\mathcal{A}}, \gamma)$ and constructing an approximation $f_P^{\mathcal{A}} : Den^{\mathcal{A}} \to Den^{\mathcal{A}}$ of f_P so that the least fixed point of $f_P^{\mathcal{A}}$ is finitely computable. This construction often takes a systematic approach which involves replacing the basic operations in the concrete semantic operator f_P by corresponding abstract operations in $f_P^{\mathcal{A}}$ (e.g., Cousot and Cousot [1992] and Nielson [1988]). Given that these abstract operations approximate the concrete operations it is generally straightforward to prove that the derived abstract semantic operator approximates the concrete semantic operator. The fundamental theorem of abstract interpretation provides the following result:

THEOREM 2.5. *Let (E, α, D, γ) be a Galois insertion, and let $\mu : E \to E$ and $\mu^{\mathcal{A}} : D \to D$ be monotonic functions such that $\mu^{\mathcal{A}}$* γ-approximates *μ. Then $lfp(\mu^{\mathcal{A}}) \propto_\gamma lfp(\mu)$.*

The *"art"* of abstract interpretation can be described as involving the following steps: (1) to choose an appropriate concrete semantics; (2) to identify a suitable notion of data description; and (3) to provide good approximations of the basic operations in the concrete semantics. Once this is done the foundation is laid for deriving, more or less automatically, a semantics-based program analysis. Applying suitable optimizations to the fixpoint algorithm used in the description domain, an analysis that is also efficient can be built essentially automatically from it [Bruynooghe 1991; Muthukumar and Hermenegildo 1992; Le Charlier and Van Hentenryck 1994]. In the case of logic programs the main step is to provide a notion of abstract substitutions and an abstract unification algorithm. Other operations include "projection" and "composition" which safely project (i.e., on a finite set of variables) and compose descriptions.

The subject of this article is centered around the practicality of: given an appropriate concrete semantics — point (1) above — and having found two or more notions of description — point (2) above — together with corresponding approximations of the basic operations in the concrete semantics — point (3) above — automatically constructing an approximation of the basic operations for a combined notion of description. Given this construction a combined analysis is derived by abstracting the concrete semantics.

Direct-Product Analysis

Let E be a concrete domain, and let $(E, \alpha_i, D_i, \gamma_i)$ $i \in \{ 1, 2 \}$ be Galois insertions. The *direct-product domain* is a quadruple $(E, \alpha_\times, D, \gamma_\times)$ where $D = D_1 \times D_2$, $\gamma_\times : D \to E$ is defined by $\lambda(d_1, d_2).\gamma_1(d_1) \sqcap_E \gamma_2(d_2)$, and $\alpha_\times : E \to D$ is defined by $\lambda e.(\alpha_1(e), \alpha_2(e))$.

The direct-product domain is not a Galois insertion. Consider for example the domain *Parity* $\times$ *Sign* (see Example 2.2). Observe that $\alpha_\times(\gamma_\times(\top, 0)) = \alpha_\times(\{0\}) = (ev, 0)$ which is in violation of Definition 2.1. However, given a function $\mu : E \to E$ and corresponding γ_i-approximations $\mu_i^A : D_i \to D_i$ for $i \in \{ 1, 2 \}$, the *direct-product function* $\mu_\times^A : D \to D$ defined by $\lambda(d_1, d_2).\ (\mu_1^A(d_1), \mu_2^A(d_2))$ is a $\gamma_\times$-approximation of μ. The direct-product function corresponds to performing the independent analyses μ_1^A and μ_2^A.

Following Cousot and Cousot [1979] we proceed to lift the direct-product domain by considering the equivalence relation induced by $\gamma_\times$. This provides a Galois insertion. Moreover, sharper analyses can be obtained by performing operations on the new *reduced-product* domain.

Reduced-Product Analysis

Let $(E, \alpha_i, D_i, \gamma_i)$, $i \in \{ 1, 2 \}$, be Galois insertions, and let $(E, \alpha_\times, D, \gamma_\times)$ be the corresponding direct-product domain. The relation $\equiv \ \subseteq D \times D$ induced by $\gamma_\times$ is defined by $d \equiv d' \Leftrightarrow \gamma_\times(d) = \gamma_\times(d')$. The *reduced-product domain* is a quadruple $(E, \alpha_\star, D_\equiv, \gamma_\star)$ where $\alpha_\star : E \to D_\equiv$ is defined by $\lambda e.\ [\alpha_\times(e)]_\equiv$ and where $\gamma_\star : D_\equiv \to E$ is defined by $\lambda[d]_\equiv.\ \gamma_\times(d)$. It is straightforward to show that the reduced-product domain is well defined and is a Galois insertion.

Let $\mu : E \to E$ be a concrete function, and let $\mu_i^A : D_i \to D_i$, $i \in \{ 1, 2 \}$, be corresponding γ_i-approximations. The *reduced-product function* $\mu_\star^A : D_\equiv \to D_\equiv$ is defined by $\lambda[d]_\equiv.\ [(\mu_1^A(d_1), \mu_2^A(d_2))]_\equiv$ where $(d_1, d_2) = \sqcap_D[d]_\equiv$, namely, the smallest

representative of the equivalence class $[d]_{\equiv}$. It is straightforward to show that the reduced-product function $\mu_{\star}^{\mathcal{A}}$ is well defined and is a $\gamma_{\star}$-approximation of μ. Moreover, in general the reduced-product function is no less precise than the direct-product function. Example 2.6 illustrates that $\mu_{\star}^{\mathcal{A}}$ is potentially more precise than $\mu_{\times}^{\mathcal{A}}$. The reduced-product function corresponds to performing the original analyses over the reduced-product domain, namely, over a domain of representatives that contain no redundant information. In practice, a constructive definition of the reduced-product function must be given. Intuitively, this involves specifying how the smallest representative of an equivalence class is to be found. Formally, one should prove (a) that a representative of the equivalence class has been chosen — *correctness*; and (b) that the representative is minimal — *optimality*.

Example 2.6. Consider the following program fragment under the initial assumption that $x = 0$. A parity analysis will start with the abstract initial invariant $x = ev$ while a sign analysis will begin with the assumption $x = 0$. After considering the first program statement the direct- and reduced-product analyses give respectively $div_{\times}^{\mathcal{A}}\langle ev, 0\rangle = \langle \top, 0\rangle$ and $div_{\star}^{\mathcal{A}}\langle ev, 0\rangle = \langle \top, 0\rangle_{\equiv}$. In the reduced-product domain $\langle \top, 0\rangle \equiv \langle ev, 0\rangle$ so after considering the second program statement the reduced-product analysis gives $inc_{\star}^{\mathcal{A}}\langle \top, 0\rangle_{\equiv} = \langle od, \hat{+}\rangle_{\equiv}$ which is more precise than the corresponding direct-product analysis $inc_{\times}^{\mathcal{A}}\langle \top, 0\rangle = \langle \top, \hat{+}\rangle$.

program fragment	direct-product analysis for x	reduced-product analysis for x
$\downarrow \{x=0\}$ $x := x \text{ div } 2$ $\downarrow \{x=0\}$ $x := x + 1$ $\downarrow \{x=1\}$	$\{ \langle ev, 0\rangle \}$ $\{ \langle \top, 0\rangle \}$ $\{ \langle \top, \hat{+}\rangle \}$	$\{ \langle ev, 0\rangle \}_{\equiv}$ $\{ \langle ev, 0\rangle \}_{\equiv}$ $\{ \langle od, \hat{+}\rangle \}_{\equiv}$

Example 2.6 demonstrates how considering the interaction between the analysis domains can sharpen precision. However, further precision may be obtained by redefining the abstract operations:

Example 2.7. Define $dec^{\mathcal{A}} : (Parity \times Sign)_{\equiv} \to (Parity \times Sign)_{\equiv}$ such that $dec^{\mathcal{A}}$ is the same as $dec_{\star}^{\mathcal{A}}$ except that $dec^{\mathcal{A}}\langle od, \hat{+}\rangle = \langle ev, \hat{+}\rangle$. This is clearly safe and provides a potentially sharper analysis.

program fragment	reduced-product analysis	reduced-product analysis with: $dec^{\mathcal{A}}\langle od, \hat{+}\rangle = \langle ev, \hat{+}\rangle$
$\downarrow \{x=1\}$ $x := x - 1$ $\downarrow \{x=0\}$	$\{ \langle od, \hat{+}\rangle \}_{\equiv}$ $\{ \langle ev, \top\rangle \}_{\equiv}$	$\{ \langle od, \hat{+}\rangle \}_{\equiv}$ $\{ \langle ev, \hat{+}\rangle \}_{\equiv}$

unification	direct-product analysis	reduced-product analysis
$\downarrow$ $\left\{ \begin{array}{l} X = U \\ Y = U \\ Z = g(U, V) \end{array} \right\}$ $\downarrow$	$\langle \{ X \}, \{ \quad \} \rangle$ $\langle \{ X, Y \}, \{ (X, Y), (Y, Z), (X, Z) \} \rangle$	$\langle \{ X \}, \{ \quad \} \rangle$ $\langle \{ X, Y \}, \{ \quad \} \rangle$

Fig. 2. Direct- and reduced-product analyses.

An Example for Logic Programs

As a simple example for logic programs, we illustrate how Søndergaard's domain for sharing and groundness analysis [Søndergaard 1986] can be represented as the reduced product of corresponding sharing and groundness domains. The resulting reduced-product analysis is equivalent to that derived from the abstract unification of Codish et al. [1991] for this domain. First we provide some preliminary definitions:

Let Var denote an enumerable set of variables and PVar $\subseteq$ Var a distinguished (enumerable) set of variables which may occur in programs. Let Sub denote the set of idempotent substitutions. Informally, a set of program variables $\{x_1, \cdots, x_n\} \subseteq$ PVar *share* or *are aliased* if in some execution of the program they may be bound to terms $t_1, \cdots, t_n$, such that $vars(t_1) \cap \cdots \cap vars(t_n) \neq \emptyset$. A program variable is *ground* if it is bound to a term t such that $vars(t) = \emptyset$. A program variable is *linear* if it is bound to a term which contains only single occurrences of variables.

Definite groundness information is described by means of a set of program variables: $D_1 = 2^{\mathrm{PVar}}$. Possible (pair) sharing information is described by symmetric binary relations on PVar: $D_2 = 2^{(\mathrm{PVar} \times \mathrm{PVar})}$. For a relation $R \in D_2$, xRy denotes that x and y are bound to terms which may share a variable; xRx denotes that x is bound to a possibly nonlinear term. For convenience we will let an arbitrary relation R on PVar denote the smallest symmetric relation which contains R.

Example 2.8. Consider the (abstract) unification $h(U, U, g(U, V)) = h(X, Y, Z)$ under the abstract substitutions $\{ X \} \in D_1$ and $\emptyset \in D_2$ specifying that X is definitely ground and that there is no possible sharing between the other variables which are definitely linear. A simple groundness analysis will determine that after the unification the variables $\{ X, Y, U \}$ will be ground. A (pair) sharing analysis which does not consider information in D_1 may determine that after unification there is at most sharing between $\{(X, Y), (Y, Z), (X, Z), (X, U), (Y, U), (Z, U), (Z, V)\}$. The *same* analyses performed on a reduced-product domain will eliminate the sharing on ground variables. Figure 2 illustrates the results of these analyses reflected on the variables $\{ X, Y, Z \}$. Such an analysis is described in Codish et al. [1991] as a formalization of Søndergaard [1986].

In the following section we present a more complex example and propose an approach to provide better precision by removing redundancies at intermediate

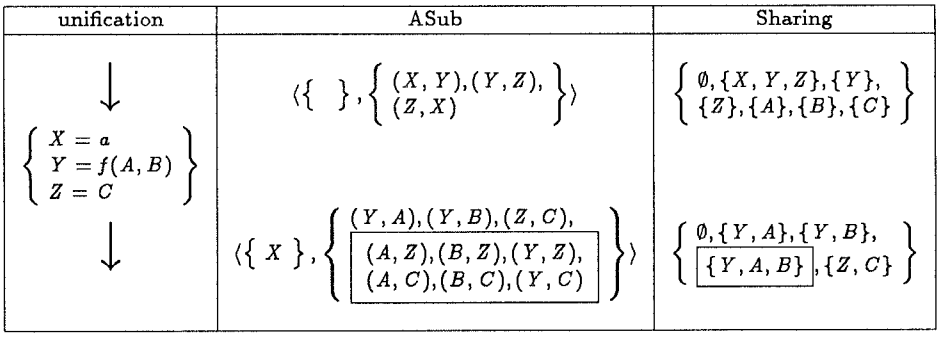

Fig. 3. Pair- and set-sharing analyses.

lower-level steps of abstract unification.

3. COMBINING SET SHARING WITH PAIR SHARING

The following illustrates the practical benefit of combining domains for a somewhat more realistic example. We consider the combination of two different sharing analyses, one over the domain ASub of Søndergaard [1986] and another one over the domain Sharing of Jacobs and Langen [1992].

The domain ASub is that described above as the reduced product $D_1 \star D_2$. An abstract substitution $(G, R) \in \mathsf{ASub}$ describes those substitutions which make (at least) all the variables in G ground and have no more *pair sharing* than specified by R. The concretization function, $\gamma_{ASub} : \mathsf{ASub} \to 2^{Sub}$, is defined by

$$\gamma_{ASub}(G,R) = \left\{ \theta \;\middle|\; \begin{array}{l} \forall (x,y) \in \mathsf{PVar}^2 : \quad (\, x \in G \Rightarrow \mathit{ground}(x\theta)\,) \quad \wedge \\ \quad (\, x \neq y \wedge \mathit{vars}(x\theta) \cap \mathit{vars}(y\theta) \neq \emptyset \Rightarrow x \; R \; y\,) \quad \wedge \\ \quad (\, x \; \not R \; x \Rightarrow \mathit{linear}(x\theta)\,) \end{array} \right\}.$$

The $\mathsf{Sharing} = 2^{2^{\mathsf{PVar}}}$ domain keeps track of *set sharing*. The concretization function is defined in terms of the occurrences of a variable U in a substitution:

$$\mathit{occs}(\theta, U) = \{X \in \mathit{dom}(\theta) \mid U \in \mathit{vars}(X\theta)\}.$$

If $\mathit{occs}(\theta, U) = V$ then θ maps the variables in V to terms which share the variable U. The concretization function $\gamma_{Sharing} : \mathit{Sharing} \to 2^{Sub}$ is defined as follows:

$$\gamma_{Sharing}(\kappa) = \{\theta \in \mathit{Sub} \mid \forall U \in \mathsf{Var}.\ \mathit{occs}(\theta, U) \in \kappa\}.$$

As mentioned before, the abstract substitution is composed of sets of program variables. Intuitively, each set in the abstract substitution containing variables $v_1, \ldots, v_n$ represents the fact that there may be one or more shared variables occurring in the terms to which $v_1, \ldots, v_n$ are bound. If a variable v does not occur in any set, then there is no variable that may occur in the terms to which v is bound, and thus those terms are definitely ground. If a variable v appears only in a singleton set, then the terms to which it is bound may contain only variables which do not appear in any other term.

The advantage of the ASub domain is that it captures information about linearity which is not captured in Sharing. In ASub whenever a term is known to be linear it is possible to infer that the variables contained in the term do not share, while in Sharing such sharing must be assumed. On the other hand, the Sharing domain is more powerful in the way groundness is propagated among variables. The reason is that Sharing not only represents when two terms possibly share, but also which variables are possibly shared and which are definitely not shared. Thus, it can represent that a set of terms share all their variables, and therefore infer the groundness of one term from the groundness of the others.

These differences make the two abstract domains incomparable in the sense that each gives better results for some programs [Bueno et al. 1994; Cortesi et al. 1992]. Several attempts have been made to enrich one domain or the other to give better results [Cortesi and Filé 1992; Sundararajan and Conery 1992]. Other combinations focus on adding other types of information while at the same time improving the sharing information [Muthukumar and Hermenegildo 1991]. These attempts all involve redefinition of the basic operations for the new domains. We propose the reduced-product domain as the straightforward way to obtain a combined analysis. Moreover, we claim that reasonable precision can be maintained without redefining the abstract unification algorithms.

It is important to note that since the reduced product is defined for two abstract domains and their abstraction (or concretization) functions, once the reduced product has been determined for them, it can be used with any abstract unification algorithm defined for the original abstract domains. In fact, in the following examples and in the evaluation of the technique given in Section 4, the abstract unification algorithms used are the respectively improved versions [Codish et al. 1991; Muthukumar and Hermenegildo 1992] of the abstract algorithms originally given for each domain.

Example 3.1. Consider the unification $p(X, Y, Z) = p(a, f(A, B), C)$ with a call pattern of the form $\{ X \mapsto U, Y \mapsto f(U, V), Z \mapsto f(U, W) \}$, namely, where X, Y, and Z are bound to linear terms which share a common variable. Figure 3 illustrates the results of abstract unification for the domains ASub and Sharing. The square boxes indicate redundant information.

The Sharing analysis indicates that the pairs (Y, Z), (Y, C), (A, Z), (A, C), (B, Z), (B, C) in the ASub analysis are redundant since they are not subsets of a set obtained by the Sharing analysis. On the other hand the ASub analysis indicates that the set $\{Y, A, B\}$ in the Sharing analysis is redundant since (A, B) is not in ASub. Hence a reduced-product analysis would result in $\{\{X\}, \{(Y, A), (Y, B), (Z, C)\}\}$ for ASub and $\{\emptyset, \{Y, A\}, \{Y, B\}, \{Z, C\}\}$ for Sharing.

In the context of this example, we adopt the following Reduce function which yields the minimal representative of an element of the reduced product given an arbitrary representative.

Definition 3.2.

$$\mathsf{Reduce} : \mathsf{ASub} \times \mathsf{Sharing} \rightarrow \mathsf{ASub} \times \mathsf{Sharing} : \langle (G, R), S \rangle \mapsto \langle (G', R'), S' \rangle$$

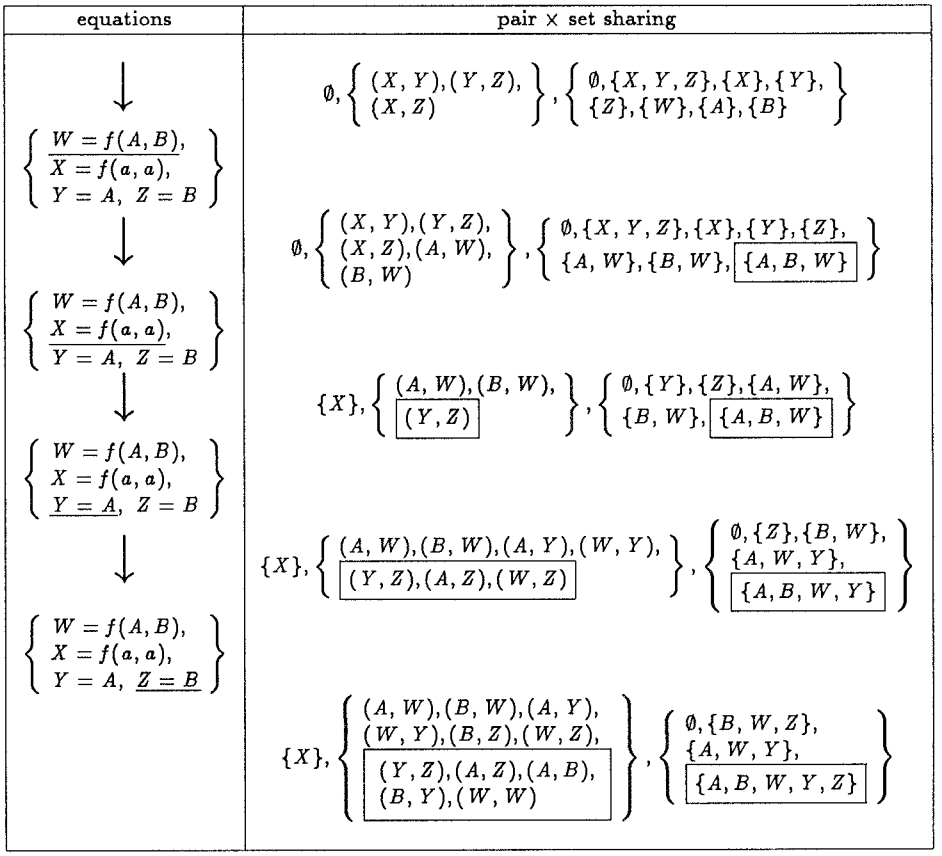

Fig. 4. Applying **Reduce** at intermediate steps.

where

$$\begin{aligned} S' &= \{s \in S \mid s \cap G = \emptyset,\ \mathsf{Pairs}(s) \subseteq R\},\\ R' &= R \cap (\underset{s \in S'}{\cup} s \times s),\\ G' &= \mathit{ground}(S'),\\ \mathsf{Pairs}(s) &= \{(X,Y) \in s \times s \mid X \neq Y\}. \end{aligned}$$

The idea is that Reduce removes redundancies from the representation of an abstract substitution while preserving its meaning. This is achieved by: (1) eliminating from S those sets which indicate sharing not present in (G,R), obtaining S'; (2) eliminating from R those pairs which indicate sharing not present in S', obtaining R'; and (3) deriving those variables which are ground according to S', obtaining G'. Proving the correctness of Definition 3.2 is not difficult. It is similar to proving correctness of the other abstract operations. Showing that it is optimal — that it provides a minimal representation — is more difficult. It involves showing that an element with more groundness or less (pair or set) sharing violates the correctness condition.

An additional gain in precision can be obtained if redundant information is removed not only after each basic operation (abstract unification), but also at intermediate steps inside the implementation of the corresponding algorithms. We illustrate this point for the combination of the ASub and Sharing domains. The abstract unification algorithms for the ASub and Sharing domains defined in Codish et al. [1991] and Jacobs and Langen [1992] both follow the same basic strategy when solving an abstract unification $E\delta$ consisting of an equation E and an abstract substitution δ, namely, first reducing the equation E to a solved form $mgu(E) = \{e_1, \ldots, e_n\}$, and then solving each of the abstract equations $e_i\delta_i$ in turn with $\delta_1 = \delta$ and δ_{i+1} the solution of $e_i\delta_i$. Consequently, the Reduce function can be applied at the intermediate steps after solving each equation in the process. Removing redundancies at intermediate steps improves both precision and efficiency of analyses. The following example demonstrates this point.

Example 3.3. Consider $E = \{p(W, X, Y, Z) = p(f(A, B), f(a, a), A, B)\}$, with $mgu(E) = \{W = f(A, B), X = f(a, a), Y = A, Z = B\}$, call substitutions $d_1 = \langle \emptyset, \{(X, Y), (Y, Z), (Z, X)\}\rangle \in$ ASub and $d_2 = \{\emptyset, \{X, Y, Z\}, \{X\}, \{Y\}, \{Z\}, \{W\}, \{A\}, \{B\}\} \in$ Sharing.

Figure 4 contains the results at the intermediate steps of the (combined) abstract unification algorithm in the case of a direct-product analysis. The sharing information in boxes is removed if the Reduce function is applied at intermediate steps in the algorithm. If intermediate redundancies are not removed then the result is less precise (indicating that Y and Z possibly share).

4. EVALUATION

This section presents and compares the analysis results obtained for the following domains:

P	ASub(pair sharing)
S	Sharing(set sharing)
SF	Sharing+Freeness
$P \star S$	reduced product
$P \star SF$	reduced product

The analyses for the domains P, S, and SF are based on the algorithms described in Codish et al. [1991], Muthukumar and Hermenegildo [1991; 1992], and Muthukumar et al. [1992] respectively. The implementations for the reduced-product analyses are provided through the Reduce function specified in Definition 3.2. In the present implementation the Reduce function is not applied at intermediate steps of the abstract unification algorithms. The analyses have been performed within the framework of the &-Prolog compiler. This framework, implemented in Prolog, is based on the abstract interpretation framework of Bruynooghe [1991], optimized with the specialized domain-independent fixpoint algorithm defined in Muthukumar and Hermenegildo [1992]. The framework is based on a collecting semantics which specifies both answer substitutions for the initial goal as well as the intermediate bindings of variables before and after each call in the body of a clause. Approximations of these intermediate bindings are relevant for many applications such as, for example, program parallelization. The choice of abstract domain is a

Table I. Program Sizes and Analysis Times

	Program sizes		Analysis times in milliseconds				
Program	# clauses	# vars	S	P	SF	P⋆S	P⋆SF
serialize	12	44	9290	839	2620	1870	2530
init-subst	14	53	569	1250	660	829	1080
map-color	13	25	4600	1040	1629	5760	2939
grammar	16	17	170	140	250	269	349
browse	38	115	51860	1609	25559	49590	29549
bid	53	98	1129	1000	1259	1429	1759
deriv	62	170	2819	2630	3119	3550	4289
rdtok	68	196	5670	4450	6879	6389	11510
read	92	344	8790	8380	9760	11069	12919
boyer	146	118	11040	3949	14600	7709	10480
peephole	155	357	20760	7990	14890	23029	25589
ann	222	594	93509	16789	44639	53269	65269

parameter of the system passed to the fixpoint algorithm which in turn calls the appropriate abstract operators. The system thus allows the comparison of precision of different analyses as well as of their relative efficiency.

The abstract operations for the domains S and SF were already supported by the existing implementation of the framework. In order to provide the results for the combined analyses, an implementation of the operations for the P domain was added. Once this was done, integrating the $P \star S$ and $P \star SF$ analyses in the system required a few additional lines of code which, each time an abstract function is called by the fixpoint algorithm, calls the corresponding abstract functions (e.g., for P and for S), performs the Reduce function over the information inferred by each analyzer, and returns the resulting information to the fixpoint algorithm.

When comparing the accuracy of the various analyses we consider a variety of criteria including information about groundness, linearity, pair sharing, and set sharing. The pair sharing of an element S of Sharing is obtained as $\{ (X, Y) \in s \times s \mid s \in S \}$. Similarly, the set sharing of an element (G, R) of ASub is obtained by considering the independent components of the transitive closure of R and removing the redundant sets as in the computation of S' in Definition 3.2.

The programs used in our evaluation are a standard set of benchmark programs. A description of the programs can be found in Codish et al. [1993]. Table I lists the programs, notes their sizes, and shows the analysis times in milliseconds (SparcStation IPC, Sicstus 2.1, native code). Size measures include the number of clauses and variables in the program. When counting clauses we note that the original programs are transformed to remove "if-then-else" and "or" structures from all clauses. Moreover, we do not count variables in facts because the analyses collect information about facts in the clauses that call them.

Example 4.1. Figure 5 illustrates the output from the set-sharing analysis S (on the left) and from the reduced-product P⋆S analysis (on the right) for one clause of the `serialize` program. The results of the analyses are indicated as comments within the text of the clause. The %S and %P notations indicate respectively information from the Sharing and ASub domains. For this clause we count 65 possibly shared sets in the Sharing analysis in contrast to the more precise 9 in the reduced-product analysis.

```
arrange([X|L],tree(T1,X,T2)) :-
  %S [[X],[X,L],[X,L,T1],[X,L,T1,T2],
  %  [X,L,T2],[X,T1],[X,T1,T2],[X,T2]
  %  ,[L],[L,T1],[L,T1,T2],[L,T2],[T1
  %  ],[T1,T2],[T2],[L1],[L2]]
 split(L,X,L1,L2),
  %S [[X],[X,L],[X,L,T1],[X,L,T1,T2],
  %  [X,L,T1,T2,L1],[X,L,T1,T2,L1,L2]
  %  ,[X,L,T1,T2,L2],[X,L,T1,L1],[X,L
  %  ,T1,L1,L2],[X,L,T1,L2],[X,L,T2],
  %  [X,L,T2,L1],[X,L,T2,L1,L2],[X,L,
  %  T2,L2],[X,L,L1],[X,L,L1,L2],[X,L
  %  ,L2],[X,T1],[X,T1,T2],[X,T2],[L,
  %  T1,T2,L1],[L,T1,T2,L1,L2],[L,T1,
  %  T2,L2],[L,T1,L1],[L,T1,L1,L2],[L
  %  ,T1,L2],[L,T2,L1],[L,T2,L1,L2],[
  %  L,T2,L2],[L,L1],[L,L1,L2],[L,L2]
  %  ,[T1],[T1,T2],[T2]]
 arrange(L1,T1),
  %S [[X],[X,L],[X,L,T1,T2,L1],[X,L,
  %  T1,T2,L1,L2],[X,L,T1,L1],[X,L,T1
  %  ,L1,L2],[X,L,T2],[X,L,T2,L2],[X,
  %  L,L2],[X,T2],[L,T1,T2,L1],[L,T1,
  %  T2,L1,L2],[L,T1,L1],[L,T1,L1,L2]
  %  ,[L,T2,L2],[L,L2],[T2]]
 arrange(L2,T2).
  %S [[X],[X,L],[X,L,T1,T2,L1,L2],[X,
  %  L,T1,L1],[X,L,T2,L2],[L,T1,T2,L1
  %  ,L2],[L,T1,L1],[L,T2,L2]]
```

```
arrange([X|L],tree(T1,X,T2)) :-
  %S [[X],[L],[T1],[T2],[L1],[L2]]
  %P [],[]
 split(L,X,L1,L2),
  %S [[X],[X,L],[L,L1],[L,L2],[T1],[T2]]
  %P [],[[L,L],[X,L],[L,L1],[L,L2]]
 arrange(L1,T1),
  %S [[X],[X,L],[L,T1,L1],[L,L2],[T2]]
  %P [],[[L,L],[L1,L1],[L,T1],[L,L1],
  %  [L,L2],[X,L],[T1,L1]]
 arrange(L2,T2).
  %S [[X],[X,L],[L,T1,L1],[L,T2,L2]]
  %P [],[[L,L],[L1,L1],[L2,L2],[X,L],
  %  [L,T1],[L,T2],[L,L1],[L,L2],
  %  [T1,L1],[T2,L2]]
```

Fig. 5. Example output for **Sharing** and reduced-product analyses.

Results

Table II indicates the total number of pairs and sets which possibly share in the S, P, and SF analyses and for the reduced-product analyses. Apart from the `init-subst` benchmark, the results for pair sharing in the reduced-product analyses are almost identical to those in the P analyses. Note that the numbers here refer to the amount of *possible* sharing, and hence more precise analyses indicate *less* sharing.

In addition, we have found that for our benchmarks, all analyses give almost the same groundness information, with one exception. In `init-subst`, the S analysis (as well as the reduced-product analyses) derives 126 definitely ground occurrences of variables in the various program points whereas the P analysis finds only 33 such occurrences. Those unrecognized ground variables are at the origin of the great difference in shared pairs between the P analysis and the $P * S$ analysis in the `init-subst` benchmark already mentioned above.

For set sharing however, the reduced-product analyses give significantly better information than the S analysis for several benchmarks. Also the time needed to perform the reduced-product analysis is often significantly better than the time needed to perform both analyses separately. Finally, we observed in our experiments that the reduced-product analysis does not improve the linearity information derived by the P analysis.

Table II. Number of Shared Pairs and Sets in Analysis Results

	no. of shared pairs					no. of shared sets				
Program	S	P	SF	P⋆S	P⋆SF	S	P	SF	P⋆S	P⋆SF
serialize	235	35	137	35	35	502	41	208	24	24
init-subst	5	72	5	5	5	5	92	5	5	5
map-color	76	74	73	73	73	108	145	101	101	101
grammar	11	11	11	11	11	5	14	5	5	5
browse	196	104	167	104	104	671	606	628	547	547
bid	11	0	0	0	0	17	0	0	0	0
deriv	0	0	0	0	0	0	0	0	0	0
rdtok	185	48	51	48	48	219	57	47	44	44
read	11	1	1	1	1	12	1	1	1	1
boyer	242	93	222	93	93	417	132	375	100	100
peephole	386	310	310	310	310	623	579	417	417	417
ann	1935	1690	1694	1690	1690	3230	6447	2543	2543	2543

Discussion

The first observation is that the results obtained for the reduced-product analyses are at least as precise as (and often more precise than) those obtained by the individual analyses. It is also interesting to note that $P{\star}SF$ does not improve the results of $P{\star}S$ in terms of sharing (although it of course provides additional freeness information). This is not surprising since freeness information provides only a restricted form of linearity information, i.e., that obtained from the knowledge that any free variable is also a linear term. Thus, $P{\star}S$ seems an excellent sharing analysis for the benchmarks used.

Although not of direct relevance to domain combination issues, the results provide an interesting comparison of the domains P, S, and SF. We observe that linearity information (present in P, not present in S, and partially present in SF) proves to be a powerful instrument for increasing the accuracy of sharing analyses.

Although it is easy to contrive examples for which the domains which capture set sharing provide a more powerful groundness propagation, it is interesting to note that in practice all of the domains provide almost identical groundness information. This is partially due to the fact that our analyses are goal dependent and because in most programs groundness typically propagates in a top-down, left-to-right direction. The `init-subst` benchmark is an exception since it contains a predicate in which groundness information propagates from right-to-left even when it is called with the most instantiated query mode. This is due to the way the program is written to take advantage of tail recursion. We believe that this phenomenon may actually show up more often in actual applications in which efficiency has been a major consideration during coding, and also when partially instantiated structures are used. Also, we expect the difference between the base domains with respect to groundness propagation and consequently with respect to both types of sharing to be more notable when performing goal-independent analyses [Barbuti et al. 1993; Codish et al. 1994a; 1994b]. It is our belief that for such analyses, combining domains is even more beneficial.

The analysis times in Table I also provide interesting insight. The time cost of the combined analysis is in many cases substantially better than the sum of the costs of the individual analyses. However, in some cases it is slightly worse. This

is the result of the interplay between different factors:

—Computing the reduced product does create extra work (which depends on the size of the inputs to the Reduce function).

—The reduced-product analysis has a "loop-merging" effect — a single pass over the program is sufficient for the combined analysis instead of two passes for the individual analyses.

—Improved accuracy in the combined analysis reduces the size of the inputs of domain-dependent operations, such as abstract unification, projection, composition, and including the reduced product.
In the case of the $P \star S$ analyses, this is a major effect due to the significant gain in accuracy. The effect is less visible in the $P \star SF$ analyses since the SF component is already more precise.

—The effects of "loop merging" can be distorted in the processing of recursive clauses where a number of fixpoint iterations are needed. This number can differ for the P, S, and combined analyses (in the $P \star S$ analysis a positive effect for `serialize, init-subst, browse, rdtok, boyer`, and a negative effect for `peephole`; in the $P \star SF$ analysis a positive effect for `browse` and a negative effect for `serialize, rdtok, peephole`). The difference in time depends very much on amount of work during the iterations.

5. CONCLUSIONS AND FUTURE WORK

We have shown how in practice it is possible to maintain precision in a combined reduced-product analysis and obtain reasonable analysis times without redefining the basic operations. We have also indicated that more precision can by achieved by breaking up the abstract operations into a sequence of smaller steps and applying the reduce function at each intermediate point. However, as shown in Example 2.7, an even sharper analysis may be obtained by redefining the abstract operations on the product domain. The less "overlapping" information the two domains have, the more likely this is. The advantage of the general approach is that proofs of correctness for the new domains are not required, and implementations can be reused. To illustrate this we have implemented a series of sharing analyses previously proposed and constructed two new ones as combinations of these. There are strong indications that our automatically combined analyses in fact compare well with other new proposals suggested in recent literature [Cortesi and Filé 1993; Sundararajan and Conery 1992] both from the point of view of efficiency and accuracy.

An important insight acquired from this work is the realization of the ease in practice of the combination process, which certainly required a much smaller amount of work than that taken by the original analyzers also implemented by us. This is of practical importance because the precision and efficiency of many analyses can be improved by combining various standard domains such as those described in this article. In many cases the improvement in efficiency is crucial for practical implementations.

Cortesi et al. [1994] have recently developed another approach for combining domains based on the so-called notion of *open products*. Their work proposes a systematic and modular approach in which the analysis designer can redefine the operations on the product domain. Using this approach the designer can focus

on the individual components of the product domain while specifying the effect of other components through so-called queries. Besides reducing the complexity of the design task, the approach is reported to reduce also the amount of analysis-specific code. However, there is a time penalty, since a direct implementation of an analysis for a particular product domain is reported to be twice as fast.

REFERENCES

BARBUTI, R., GIACOBAZZI, R., AND LEVI, G. 1993. A general framework for semantics-based bottom-up abstract interpretation of logic programs. *ACM Trans. Program. Lang. Syst. 15,* 1, 133–181.

BRUYNOOGHE, M. 1991. A practical framework for the abstract interpretation of logic programs. *J. Logic Program. 10,* 2 (Feb.), 91–124.

BRUYNOOGHE, M. AND BOULANGER, D. 1994. Abstract interpretation for (constraint) logic programming. In *Constraint Programming,* B. Mayoh, E. Tyugu, and J. Penjam, Eds. NATO Advanced Science Institutes Series, vol. F/131. Springer-Verlag, Berlin, 228–258.

BUENO, F., GARCÍA DE LA BANDA, M., AND HERMENEGILDO, M. 1994. Effectiveness of global analysis in strict independence-based automatic program parallelization. In *Proceedings of the International Symposium on Logic Programming.* MIT Press, Cambridge, Mass.

CODISH, M., DAMS, D., AND YARDENI, E. 1994a. Bottom-up abstract interpretation of logic programs. *J. Theor. Comput. Sci. 124,* 93–125.

CODISH, M., DAMS, D., AND YARDENI, E. 1991. Derivation and safety of an abstract algorithm for groundness and aliasing analysis. In *Proceedings of the 8th International Conference on Logic Programming* (Paris, France). MIT Press, Cambridge, Mass., 79–93.

CODISH, M., GARCÍA DE LA BANDA, M., BRUYNOOGHE, M., AND HERMENEGILDO, M. 1994b. Goal dependent vs. goal independent analysis of logic programs. In *Proceedings of the 5th International Conference on Logic Programming and Automated Reasoning.* Lecture Notes in Artificial Intelligence, vol. 822. Springer-Verlag, Berlin, 305–320.

CODISH, M., MULKERS, A., BRUYNOOGHE, M., GARCÍA DE LA BANDA, M., AND HERMENEGILDO, M. 1993. Improving abstract interpretations by combining domains. In *Proceedings of the ACM Symposium on Partial Evaluation and Semantics-based Program Transformation.* ACM Press, New York, 194–205.

CORTESI, A. AND FILÉ, G. 1993. Comparison and design of abstract domains for sharing analysis. In *Proceedings of the 8th Italian Conference on Logic Programming, GULP'93* (Gizzeria Lido), D. Saccà, Ed. Institut d'Investigació en Intel·ligència Artificial. CSIC.

CORTESI, A. AND FILÉ, G. 1992. Freeness computation in abstract interpretation. Rapporto Interno n.2/92 (March), Dip. di Matematica Pura e Applicata, Universitá di Padova, Italia.

CORTESI, A., FILÉ, G., AND WINSBOROUGH, W. 1992. Comparison of abstract interpretations. In *Proceedings of the 19th International Colloquium on Automata, Languages, and Programming.* Lecture Notes in Computer Science, vol. 623. Springer-Verlag, Berlin.

CORTESI, A., LE CHARLIER, B., AND VAN HENTENRYCK, P. 1994. Combinations of abstract domains for logic programming. In *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oreg.). ACM Press, New York, 227–239.

COUSOT, P. AND COUSOT, R. 1992. Abstract interpretation and application to logic programs. *J. Logic Program. 13,* 2 and 3 (July), 103–179.

COUSOT, P. AND COUSOT, R. 1979. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM Symposium Principles of Programming Languages* (San Antonio,

Tex.). ACM, New York, 269–282.

COUSOT, P. AND COUSOT, R. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages* (Los Angeles, Calif.). ACM, New York, 238–252.

DEBRAY, S. K., (Ed.). 1992. Special issue: Abstract interpretation. *J. Logic Program. 13,* 2 and 3 (July).

HERMENEGILDO, M. AND GREENE, K. 1990. &-Prolog and its performance: Exploiting independent and-parallelism. In *Proceedings of the 7th International Conference on Logic Programming* (Jerusalem, Israel). MIT Press, Cambridge, Mass., 253–268.

HORIUCHI, K. 1992. Less abstract semantics for abstract interpretation of FGHC programs. In *Proceedings of the International Conference on Fifth Generation Computer Systems* (Tokyo, Japan). ICOT, Tokyo, 897–906.

JACOBS, D. AND LANGEN, A. 1992. Static analysis of logic programs for independent and-parallelism. *J. Logic Program. 13,* 2 and 3 (July), 291–314.

JANSSENS, G. AND BRUYNOOGHE, M. 1992. Deriving descriptions of possible values of program variables by means of abstract interpretation. *J. Logic Program. 13,* 2 and 3 (July), 205–258.

JONES, N. D. AND SØNDERGAARD, H. 1987. A semantic-based framework for the abstract interpretation of Prolog. In *Abstract Interpretation of Declarative Languages*, S. Abramsky and C. Hankin, Eds. Ellis Horwood, Chichester, U.K., 123–142.

LE CHARLIER, B. AND VAN HENTENRYCK, P. 1994. Experimental evaluation of a generic abstract interpretation algorithm for prolog. *ACM Trans. Program. Lang. Syst. 16,* 1, 35–101.

MUTHUKUMAR, K. AND HERMENEGILDO, M. 1992. Compile-time derivation of variable dependency using abstract interpretation. *J. Logic Program. 13,* 2 and 3 (July), 315–347.

MUTHUKUMAR, K. AND HERMENEGILDO, M. 1991. Combined determination of sharing and freeness of program variables through abstract interpretation. In *Proceedings of the 8th International Conference on Logic Programming* (Paris, France). MIT Press, Cambridge, Mass., 49–63.

MUTHUKUMAR, K., GARCÍA DE LA BANDA, M., AND HERMENEGILDO, M. 1992. Sharing and freeness analysis of logic programs using abstract interpretation. Tech. Rep. (Nov.), T. U. of Madrid (UPM), Facultad Informática UPM, Madrid, Spain.

NIELSON, F. 1988. Strictness analysis and denotational abstract interpretation. *Inf. Comput. 76,* 29–92.

SØNDERGAARD, H. 1986. An application of abstract interpretation of logic programs: Occur check reduction. In *ESOP'86 Proceedings European Symposium on Programming*, B. Robinet and R. Wilhelm, Eds. Lecture Notes in Computer Science, vol. 213. Springer-Verlag, New York, 327–338.

SUNDARARAJAN, R. AND CONERY, J. 1992. An abstract interpretation scheme for groundness, freeness and sharing analysis of logic programs. In *Conference on Foundations of Software Technology and Theoretical Computer Science.* Lecture Notes in Computer Science, vol. 652. Springer-Verlag, New York, 203–216.